Information and Computation 270 (2020) 104461

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Absent words in a sliding window with applications ☆

Maxime Crochemore [a,b], Alice Héliou [c], Gregory Kucherov [b,d], Laurent Mouchard [e], Solon P. Pissis [f,*], Yann Ramusat [g]

[a] Department of Informatics, King's College London, London, UK
[b] CNRS & Université Paris-Est, Champs-sur-Marne, France
[c] LIX, École Polytechnique, CNRS, INRIA, Université Paris-Saclay, Palaiseau, France
[d] SkolTech, Moscow, Russia
[e] University of Rouen, LITIS EA 4108, TIBS, Rouen, France
[f] Centrum Wiskunde & Informatica, 1098 XG Amsterdam, the Netherlands
[g] DI ENS, ENS, CNRS, PSL Research University & INRIA, Paris, France

ARTICLE INFO

Article history:
Received 13 May 2018
Received in revised form 8 January 2019
Accepted 28 January 2019
Available online 4 September 2019

Keywords:
Algorithms on strings
Pattern matching
Absent words
Forbidden words
Online algorithms

ABSTRACT

An *absent word* of a word y is a word that does not occur in y. It is then called *minimal* if all its proper factors occur in y. In fact, minimal absent words (MAWs) provide useful information about y and thus have several applications. In this paper, we propose an algorithm that maintains the set of MAWs of a fixed-length window sliding over y online. Our algorithm represents MAWs through nodes of the suffix tree. Specifically, the suffix tree of the sliding window is maintained using modified Senft's algorithm (Senft, 2005), itself generalizing Ukkonen's online algorithm (Ukkonen, 1995). We then apply this algorithm to the approximate pattern-matching problem under the Length Weighted Index distance (Chairungsee and Crochemore, 2012). This results in an online $\mathcal{O}(\sigma|y|)$-time algorithm for finding approximate occurrences of a word x in y, $|x| \leq |y|$, where σ is the alphabet size.

1. Introduction

An *absent word* of a word y is a word that does not occur in y. It is then called *minimal* if all its proper factors occur in y. This notion was first studied in [4] under the name of minimal forbidden words. The set of minimal absent words (hereafter MAWs), sometimes called *antidictionary*, provides useful information about y. Among possible applications, MAWs are used for data compression [14,27], music information retrieval [9] or for alignment-free sequence comparison through a word distance based on the symmetric difference of MAW sets [6]. MAWs are also meaningful in molecular biology: for instance, three MAWs of the human genome were predicted to play a functional role in a coding region of Ebola virus genomes [31]. Other works dealing with MAWs in genomic sequences include [19,21,33,10,1,20].

A tight upper bound on the number of MAWs of a word y of length n over an alphabet of size σ is known to be $\mathcal{O}(\sigma n)$ [13,22,7]. It was also shown that the set of all MAWs of y is sufficient to uniquely reconstruct y [13,15]. Several linear-time and linear-space algorithms have been proposed to compute the set of MAWs for constant-sized [13,26,17,5,2,3] or integer [16,8] alphabets. These algorithms are based on text indexing data structures such as suffix tree, suffix array or

☆ A preliminary version of this paper was presented at the 21st International Symposium on Fundamentals of Computation Theory (FCT 2017) [12].

* Corresponding author.
E-mail addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), alice.heliou@polytechnique.org (A. Héliou), gregory.kucherov@univ-mlv.fr (G. Kucherov), laurent.mouchard@univ-rouen.fr (L. Mouchard), solon.pissis@cwi.nl (S.P. Pissis), yann.ramusat@ens.fr (Y. Ramusat).

https://doi.org/10.1016/j.ic.2019.104461
0890-5401/

directed acyclic word graph (DAWG and Factor automaton). Algorithms for computing the set of shortest absent words have been proposed as well (see [33] and references therein). Notice that MAWs form a superset of shortest absent words.

In this paper, we study the computation of MAWs in the *online* setting. Dynamically maintaining the set of MAWs online for a growing word has been studied in [25], where a linear-time algorithm has been proposed for this problem. Here we consider the problem of maintaining the set of MAWs in a fixed-length suffix of a growing word. In other words, we want to maintain online the set of MAWs within a fixed-length window sliding over word y.

Our motivation comes from the problem of online pattern matching under the distance called *Length Weighted Index (LWI)*, introduced in [6]. LWI is based on the symmetric difference of MAW sets. It has been applied to design an $\mathcal{O}(m+n)$-time and $\mathcal{O}(m+n)$-space algorithm for alignment-free comparison of two sequences of length m and n for constant-sized [10] or integer [8] alphabets. Different measures based on LWI have also been studied for alignment-free sequence comparison and phylogeny reconstruction [28]. To maintain the LWI across y for a word x, we need to compute the set of MAWs for a sliding window of size $m = |x|$ of y. To the best of our knowledge, this problem has not been addressed yet.

This problem can be viewed as a variant of the classic approximate pattern-matching problem in which the distance of the *pattern* of length m to a factor of length m of the *text* is the LWI distance. Note that LWI verifies metric conditions [7]. The problem of approximate pattern matching admits many different formulations and has been the subject of many works (see [18,11,24]). However, most of these works define the similarity measure based on certain types of *errors* (such as letter substitutions, insertions, deletions, or letter swaps). Here we adopt a different definition and use the LWI distance, which is based on MAWs, a type of negative information. Note finally that the online version of pattern matching is a long-standing problem having numerous efficient solutions [11]; however its approximate counterpart is more difficult to handle, and most of the work on approximate pattern matching is done in the offline framework. We refer the interested reader to [23] for results on online approximate pattern matching.

Our contributions can be summarized as follows. We present the first algorithm to maintain the set of MAWs for a sliding window. For a window of size m and a word y of length n over an alphabet of size σ, our algorithm performs $\mathcal{O}(\sigma n)$ insert and delete operations on the dynamically changing set of MAWs. The algorithm is based on Senft's algorithm [29] for maintaining the suffix tree for a sliding window extended by additional information encoding the set of MAWs. As a side result, restricted to just appending a new letter to the right, our algorithm can been seen as a simplification of the algorithm of [25] that uses a parallel construction of two suffix trees. With a careful implementation of an extended dynamic suffix tree, our algorithm requires $\mathcal{O}(\sigma n)$ time overall using $\mathcal{O}(\sigma m)$ space. We then apply this algorithm to the approximate pattern-matching problem under the LWI distance for a word x of length m (the pattern) against every window of size m of y (the text). This results in an online $\mathcal{O}(\sigma n)$-time algorithm. Thus the time complexity of the pattern-matching algorithm does not depend on m. This yields the first online pattern-matching algorithm based on some form of negative information (MAWs) for the comparison.

1.1. Definitions and notation

Let $y = y[0]y[1]\cdots y[n-1]$ be a *word* of length $n = |y|$ over a finite ordered *alphabet* of size $\sigma = |\Sigma|$. We denote by $y[i\,..\,j] = y[i]\cdots y[j]$ the *factor* of y whose occurrence *starts* at position i and *ends* at position j on y, and by ε the *empty word*, the word of length 0. The set of all possible words on Σ (including the empty word) is denoted by Σ^*. A *prefix* of y is a factor that starts at position 0 ($y[0\,..\,j]$) and a *suffix* is a factor that ends at position $n-1$ ($y[i\,..\,n-1]$). A factor x of y is *proper* if $x \neq y$.

Let u be a non-empty word. An integer p, $0 < p \leq |u|$, is called a *period* of u if $u[i] = u[i+p]$, for $i = 0, 1, \ldots, |u| - p - 1$. For every word u and every natural number k, we define the kth *power* of the word u, denoted by u^k, by $u^0 = \varepsilon$ and $u^k = u^{k-1}u$, for $k = 1, 2, \ldots, n$.

Let x be a word of length $m \leq n$. We say that there exists an *occurrence* of x in y when x is a factor of y. Opposingly, we say that the word x is an *absent* word of y if it does not occur in y. We consider absent words of length at least 2 only. An absent word x of length m, $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y. This is equivalent to saying that a minimal absent word (MAW) of y is of the form aub, $a, b \in \Sigma, u \in \Sigma^*$, such that au and ub are factors of y but aub is not. We denote by $M(y)$ is the set of MAWs of y. We can easily see that, if x is a MAW of y, then $2 \leq |x| \leq |y| + 1$. Note that $|x| = |y| + 1$ if and only if $y = a^{|y|}$ for some $a \in \Sigma$.

Example 1. *Let $y = ABAACA$. Its factors of lengths 1 and 2 are A, B, C, AA, AB, AC, BA, and CA. All MAWs of y can be obtained by combining those factors:*

$$M(ABAACA) = \{BB, BC, CB, CC, AAA, AAB, BAB, BAC, CAA, CAB, CAC\}.$$

Let U and V be two sets. We denote by $U \triangle V$ their symmetric difference, that is, $U \triangle V = (U \setminus V) \cup (V \setminus U)$. The Length Weighted Index (LWI) is a distance on Σ^* based on the set $M(x) \triangle M(y)$ [6]. It is defined by

$$\mathsf{LWI}(x, y) = \sum_{w \in M(x) \triangle M(y)} \frac{1}{|w|^2}.$$

It has been proved in [7] that LWI is a metric on Σ^*.

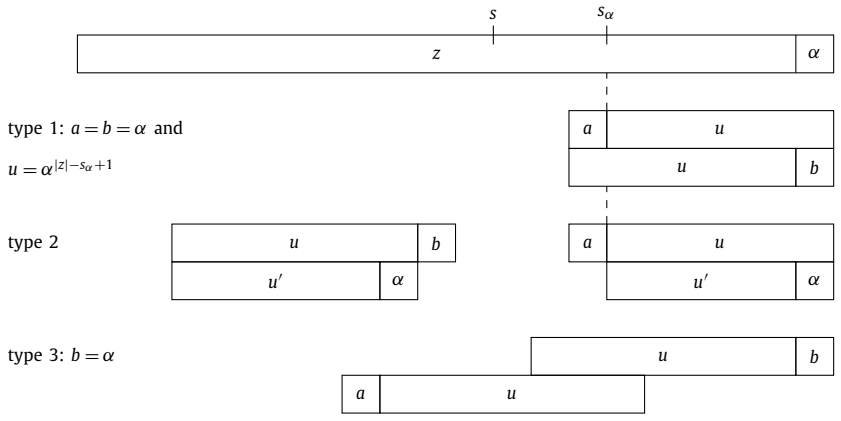

Fig. 1. Illustration of the three different types of MAWs that are added when letter α is appended to z.

2. Combinatorial results

In this section we consider a word z of fixed length m over an alphabet Σ of size σ and denote by $M(z)$ its set of MAWs. The word z represents the current window on word y used later in the algorithm of Section 3. Shifting the window by one position amounts to appending a letter to the right and deleting the leftmost letter. We first discuss the updates to be made to the set of MAWs as a result of these two operations. We then prove bounds on the number of updates of the set of MAWs caused by a window shift.

2.1. Updates caused by appending a letter to the right

We denote by $M(z)|\alpha$, $\alpha \in \Sigma$, the operation of update of the set $M(z)$ after appending the letter α to a possibly empty word z. In other words, $M(z)|\alpha$ transforms $M(z)$ into $M(z\alpha)$. Below we derive bounds on the number of insertions/deletions of MAWs caused by this operation. Some of these results have already been obtained in [25] and we briefly present them for completeness.

Let s be the starting position of the longest suffix of z that repeats in z ($s = |z|$ if this suffix is empty). Let s_α be the starting position of the longest suffix that occurs elsewhere in z followed by α ($s_\alpha = |z|$ if this suffix is empty). Note that $s \le s_\alpha$, see Fig. 1.

The next two lemmas state bounds on the number of inserted and deleted MAWs caused by $M(z)|\alpha$.

Lemma 2. *$M(z)|\alpha$ deletes exactly one MAW from $M(z)$ which is $z[s_\alpha - 1\,..\,|z| - 1]\alpha$.*

Proof. Let $w = aub$, $a, b \in \Sigma$ and $u \in \Sigma^*$, be a MAW to be removed. This means that aub is absent in z but present in $z\alpha$. Thus $b = \alpha$ and au is a suffix of z that does not occur followed by α in z. The word $ub = u\alpha$ is also present in z, so u is a suffix of z that occurs in z followed by α. Then the starting position of the suffix occurrence of u in z is s_α and $w = z[s_\alpha - 1\,..\,|z| - 1]\alpha$. □

To establish an upper bound on the number of MAWs added by operation $M(z)|\alpha$, we first divide the new MAWs of the form aub, $a, b \in \Sigma$ and $u \in \Sigma^*$, into three types (see also Fig. 1):

1. au and ub are both absent in z.
2. au is absent in z and ub is present in z.
3. au is present in z and ub is absent in z.

Lemma 3. *There are at most one MAW of type 1, σ MAWs of type 2, and $(s_\alpha - s)(\sigma - 1)$ MAWs of type 3 added by operation $M(z)|\alpha$.*

Proof. Consider a newly created MAW $w = aub$, $a, b \in \Sigma$ and $u \in \Sigma^*$. Let w be of type 1, that is neither au nor ub occurs in z. Then they are both suffixes of $z\alpha$, and since they have same length, are equal. This implies that u is both a prefix and a suffix of $ub = u\alpha$. Thus, the latter has period 1, w is of the form $\alpha^{|w|}$, and $u = \alpha^{|w|-2}$. However, $u\alpha$ is then absent in z. Therefore, $\alpha^{|w|-3}$ is the longest repeated suffix of z that occurs followed by α in z. Consequently, $|w| = |z| - s_\alpha + 3$.

Let w be of type 2, that is, ub occurs in z and au occurs in $z\alpha$ but not in z. Then au is a suffix of $z\alpha$ and u can be written $u'\alpha$. As ub occurs in z, u' is a suffix of z that occurs in z followed by α. Moreover, since $au = au'\alpha$ does not occur

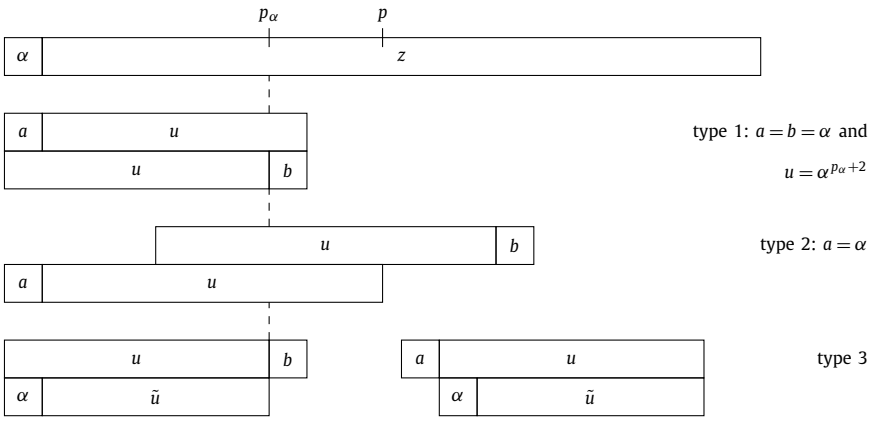

Fig. 2. Illustration of the three different types of MAWs that are deleted when removing α, the letter before z.

in z, u' is the longest suffix of z that occurs in z followed by α, therefore its starting position as a suffix is s_α. Letter b can be any letter of the alphabet of z that occurs after an occurrence of u in z. Consequently, there are at most σ such MAWs.

Let w be of type 3, that is, au occurs in z and ub occurs in $z\alpha$ but not in z. This implies that $b = \alpha$, u is a suffix of z not preceded by a, and au occurs elsewhere in z. Since no occurrence of u in z is followed by α, we have that the starting position k of u as a suffix satisfies $s \le k < s_\alpha$. Therefore, there are at most $s_\alpha - s$ possible words u and for each of them, there are at most $\sigma - 1$ possibilities for the letter a to obtain a MAW. Consequently, there are at most $(s_\alpha - s)(\sigma - 1)$ such MAWs. □

Lemma 3 implies that at a single shift of a window of size m, we may have to handle $\mathcal{O}(\sigma m)$ new MAWs. However, the total number of newly created MAWs for an entire word y of length n gets amortized to $\mathcal{O}(\sigma n)$ in an online computation.

Proposition 4 *([25]). Starting with the empty word z, and applying n times the operation $M(z)|\alpha$ leads to a total of $\mathcal{O}(\sigma n)$ inserted/deleted MAWs.*

Proof. The number of MAWs of the whole word of length n is in $\mathcal{O}(\sigma n)$ [13]. As stated by Lemma 2, at most one MAW can be deleted by each operation. Thus, the total number of insertions/deletions is $\mathcal{O}(\sigma n)$. □

2.2. Updates caused by removing the leftmost letter

We denote by $M(\alpha z) \to M(z)$, $\alpha \in \Sigma$, the operation of transforming $M(\alpha z)$ into $M(z)$. This operation is inverse to the one considered in the previous section. We now focus on the longest repeated prefix instead of the longest repeated suffix.

Let p be the ending position of the longest repeated prefix of z and by p_α the ending position of the longest prefix of z that occurs elsewhere in z preceded by α. We set them to 0 when the prefixes are empty. Note that $p_\alpha \le p$. Similar to Lemma 2, removing a letter from the left creates exactly one MAW.

Lemma 5. *Operation $M(\alpha z) \to M(z)$ creates exactly one MAW which is $\alpha z[0\,..\,p_\alpha + 1]$.*

Similar to Section 2.1, we distinguish three types of MAWs to be deleted by the operation:

1. au and ub are both absent in z.
2. au is absent in z and ub present in z.
3. ub is absent in z and au present in z.

MAWs of type 1, 2, and 3 are counterparts to MAWs of respectively type 1, 3, and 2 considered in Section 2.1, see Fig. 2 for illustration. The following result is similar to Lemma 3.

Lemma 6. *There are at most one MAW of type 1, $(\sigma - 1)(p - p_\alpha)$ MAWs of type 2, and σ MAWs of type 3 to be deleted by operation $M(\alpha z) \to M(z)$.*

2.3. Updates caused by sliding a window

We now focus on our main issue of maintaining the MAWs for a sliding window. For $m < n$ and for all i, $0 \le i \le n-m$, we consider the window $y[i\,..\,i+m-1]$ and define the following positions.

- s_i the starting position of its longest repeated suffix,
- $\tilde{s}_i$ the starting position of its longest suffix that occurs followed by $y[i+m]$,
- ss_i the starting position of its longest suffix that is a power,
- p_i the ending position of its longest repeated prefix,
- $\tilde{p}_i$ the ending position of its longest prefix that occurs preceded by $y[i-1]$,
- pp_i the ending position of its longest prefix that is a power.

The following lemma shows that explicitly enumerating the MAW sets for all windows would take the prohibitive $\mathcal{O}(\sigma nm)$ time.

Lemma 7. $\sum_{i=0}^{n-m} |M(y[i\,..\,i+m-1])|$ *is in* $\mathcal{O}(\sigma nm)$ *and this bound is tight.*

Proof. Let $\Sigma = \{\mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_3, \ldots, \mathsf{A}_\sigma\}$ with $\sigma \ge 2$. For any m, n, with $\sigma \ll m \le n$ and n multiple of m, consider the word:

$$y = (\mathsf{A}_1^r \mathsf{A}_2 \mathsf{A}_1^k \mathsf{A}_3 \mathsf{A}_1^k \cdots \mathsf{A}_i \mathsf{A}_1^k \mathsf{A}_{i+1} \mathsf{A}_1^k \cdots \mathsf{A}_\sigma \mathsf{A}_1^k)^{\frac{n}{m}},$$

where $k = \left\lfloor \frac{m}{\sigma-1} \right\rfloor - 1$ and $r = m - (\sigma-1)(k+1)$. Note that y is of length n.

Consider a factor w of y of length m. One of its prefixes is of the form $\mathsf{A}_1^p \mathsf{A}_a$ with $0 \le p \le k+r$, $2 \le a \le \sigma$, and one of its suffixes is of the form $\mathsf{A}_b \mathsf{A}_1^s$ with $0 \le s \le k+r$, $2 \le b \le \sigma$. By construction of y we have $k \le p+s \le k+r$. Then, all the words $\{\mathsf{A}_1^j \mathsf{A}_\ell, 0 \le j \le k, 2 \le \ell \le \sigma\} \setminus \{\mathsf{A}_1^j \mathsf{A}_a | p < j \le k\}$ occur in w. Similarly, all the words $\{\mathsf{A}_i \mathsf{A}_1^j, 0 \le j \le k, 2 \le i \le \sigma\} \setminus \{\mathsf{A}_b \mathsf{A}_1^j | s < j \le k\}$ occur in w.

Define $F = \{\mathsf{A}_i \mathsf{A}_1^j \mathsf{A}_\ell | 0 \le j \le k, 2 \le i \le \sigma, 2 \le \ell \le \sigma\}$ and subsets $P(w), S(w) \subseteq F$ by

$$P(w) = \{\mathsf{A}_i \mathsf{A}_1^j \mathsf{A}_a | p < j \le k, 2 \le i \le \sigma\},\ S(w) = \{\mathsf{A}_b \mathsf{A}_1^j \mathsf{A}_\ell | s < j \le k, 2 \le \ell \le \sigma\}.$$

All proper factors of words in $N(w) = F - P(w) - S(w)$ occur in w. Thus, words in $N(w)$ are either present in w or are MAWs of w. Observe that $|F| = (k+1)(\sigma-1)(\sigma-1)$, $|P(w)| = (\sigma-1)(k-p)$ and $|S(w)| = (\sigma-1)(k-s)$. Since $k \le p+s \le k+r$, we have $|P(w)| + |S(w)| \le k(\sigma-1)$. Consequently, $|N(w)| \ge (k+1)(\sigma-1)(\sigma-2) + \sigma - 1$.

By construction of y there are at most $\sigma-2$ words of $N(w)$ that occur in w. Then

$$|M(w)| \ge (k+1)(\sigma-1)(\sigma-2) + 1 = \left\lfloor \frac{m}{\sigma-1} \right\rfloor (\sigma-1)(\sigma-2) + 1 \ge (m-\sigma)(\sigma-2) + 1.$$

Therefore, $|M(w)| = \Omega(\sigma m)$ and since $|M(w)| = \mathcal{O}(\sigma m)$, we have $|M(w)| = \Theta(\sigma m)$. Summing up over $(n-m)$ windows $w = M(y[i\,..\,i+m-1])$, the lemma is proved. □

However, the following result shows that the total number of updates (deleted and inserted MAWs) over all windows is only $\mathcal{O}(\sigma n)$.

Theorem 8. $\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1]) \triangle M(y[i+1\,..\,i+m])|$ *is in* $\mathcal{O}(\sigma n)$.

Proof. Consider the set $M(y[i\,..\,i+m-1]) \triangle M(y[i\,..\,i+m])$ for some $0 \le i < n-m$, corresponding to the updates caused by appending $y[m]$ to the window $y[i\,..\,i+m-1]$. From Lemmas 2 and 3, we have

$$|M(y[i\,..\,i+m-1]) \triangle M(y[i\,..\,i+m])| \le (\tilde{s}_i - s_i)(\sigma-1) + \sigma + 2.$$

Then,

$$\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1]) \triangle M(y[i\,..\,i+m])| \le \sum_{i=0}^{n-m-1} (\tilde{s}_i - s_i)(\sigma-1) + n\sigma + 2n.$$

Observe that $\tilde{s}_i \le s_{i+1} \le \tilde{s}_i + 1$ and $s_i \le \tilde{s}_i$, therefore

$$0 \le \sum_{i=0}^{n-m-1} (\tilde{s}_i - s_i) = \sum_{i=0}^{n-m-1} \tilde{s}_i - \sum_{i=0}^{n-m-1} s_i = \tilde{s}_{n-m-1} - s_0 + \sum_{i=0}^{n-m-2} (\tilde{s}_i - s_{i+1}) \le n.$$

Then $\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1]) \triangle M(y[i\,..\,i+m])| \leq 2n\sigma + n$.

Now consider the set $M(y[i\,..\,i+m]) \triangle M(y[i+1\,..\,i+m])$ corresponding to the updates caused by deleting $y[i]$ from the window $y[i\,..\,i+m]$. From Lemmas 5 and 6, we obtain a similar inequality:

$$\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m]) \triangle M(y[i+1\,..\,i+m])| \leq 2n\sigma + n.$$

Since the number of updates in shifting from $y[i\,..\,i+m-1]$ to $y[i+1\,..\,i+m]$ is bounded by the sum of the two bounds above, we obtain the theorem. □

3. Algorithm for maintaining the MAWs in a sliding window

The *suffix tree* T of a non-empty word w of length n is a compact trie representing all suffixes of w. The nodes of the trie which become nodes of the suffix tree (i.e., branching nodes and leaves) are called *explicit* nodes, the other nodes are called *implicit*. We use $L(v)$ to denote the path-label of a node v, i.e., the concatenation of the edge labels along the path from the root to v. Node v is a *terminal* node if and only if $L(v) = w[i\,..\,n-1]$, $0 \leq i < n$, in which case v is labelled with position i. The *suffix link* of a node v with path-label $L(v) = \alpha s$ is a pointer to the node path-labelled s, where $\alpha \in \Sigma$ is a single letter and s is a word. The suffix link of v exists if v is a non-root internal node of T. For more about suffix trees we refer e.g. in [11].

Our algorithm below relies on Senft's online construction algorithm of the suffix tree for a sliding window [30] that is itself based on Ukkonen's online construction algorithm of the suffix tree [32].

3.1. An overview of Senft's algorithm

The algorithm by Ukkonen constructs the suffix tree of a word of length n online in $\mathcal{O}(n)$ time over a constant-sized alphabet by processing the word from left to right. To adapt it to a sliding window with amortized constant time per one window shift, two additional issues need to be resolved: (i) how to delete the leftmost letter of a window, and (ii) how to maintain edge labels under window shifts.

3.1.1. Deleting the leftmost letter

Consider the longest repeated prefix of the current window. When the leftmost letter is deleted, all prefixes that are longer than this prefix have to be removed from the tree, and the longest repeated prefix and all shorter prefixes should be kept in the tree. To remove these prefixes, we proceed as follows.

- If the longest repeated prefix corresponds to an explicit node, this node is the parent of the leaf corresponding to the entire window. This leaf is deleted, together with the incoming edge. If this node has only one child remaining, the node is deleted and the two edges are merged.
- If the longest repeated prefix corresponds to an implicit node, it is equal to the longest repeated suffix, and the node is located on the edge leading to the leaf corresponding to the entire window. This leaf is then relabelled with the starting position of what was the longest repeated suffix and its incoming edge is labelled accordingly.

3.1.2. Maintaining edge labels

We apply batch update of edge labels [29]. Assume by induction that all edge labels are correctly positioned relative to the current window. For the next m shifts of the window, we still maintain the same relative positioning of edge labels. After m shifts, edge labels are recomputed by a bottom-up traversal of the tree. Since m shifts create at most $2m$ nodes, the amortized time spent on one shift is $\mathcal{O}(1)$.

3.2. Our algorithm

Consider a word y of length n over an alphabet Σ of size σ. Our goal is to maintain the set of MAWs for a sliding window of size m. That is, for all successive $i \in [0, \ldots, n-m]$, we want to compute $M_m(i) = M(y[i\,..\,i+m-1])$.

For a word z, by $\Sigma(z)$ we denote the alphabet of z and by $V(z)$ the set of explicit nodes in the suffix tree of z. Consider a mapping $f : M(z) \to \Sigma(z) \times V(z)$ defined by $f(aub) = (a, v_{ub})$, where $a \in \Sigma$ and v_{ub} is either the explicit node corresponding to the factor ub or the immediate explicit descendant node if this node is implicit.

Lemma 9. *Mapping f is an injection.*

Proof. Let $w, w' \in M(z)$, $w \neq w'$, $w = aub$ and $w' = a'u'b'$, with $a, b, a', b' \in \Sigma(z)$ and $u, u' \in \Sigma(z)^*$.

Suppose that $f(w) = f(w')$, then $a = a'$ and $v_{ub} = v_{u'b'}$. Thus ub and $u'b'$ are distinct prefixes of the factor corresponding to v_{ub}, consequently one is prefix of the other, without loss of generality ub is prefix of $u'b'$. Then aub is a prefix of $au'b'$,

this is impossible as they are both MAWs of z. Thus two distinct elements of $M(z)$ cannot share the same image by f, so f is an injection. □

Lemma 9 allows us to represent all MAWs by storing a set of letters in each explicit node of the tree. We will call this set the *maw*-set. Moreover, a letter a in the *maw*-set will be *tagged* if and only if u corresponds to an implicit node in the tree. Observe that a can become tagged only when u is a repeated suffix of y. This is because factors au and ub define distinct occurrences of u, and the occurrence of au must be a suffix, otherwise u would be followed by two distinct letters and would then be an explicit node. Besides *maw*-sets, each explicit node will store another set of letters: the set of all letters preceding the occurrences of the factor corresponding to the node.

By induction, assume we are at position i, the suffix tree $T_m(i)$ for $y[i\,..\,i+m-1]$ has been built and the set of MAWs $M_m(i)$ has been computed. We now explain how to update $T_m(i)$ and $M_m(i)$ to obtain $T_m(i+1)$ and $M_m(i+1)$. The tree is updated based on Senft's algorithm, by first appending a letter to the right of the current window and then deleting the leftmost letter. The set of MAWs is updated using Lemmas 2, 3 and 5, 6 respectively. The algorithm will maintain positions s_i, p_i, $\tilde{s}_i$, $\tilde{p}_i$, ss_i, pp_i as defined in Section 2.3. We store the leaf nodes in a list so that the last created leaf and the "oldest" leaf currently in the tree can be accessed in constant time.

3.2.1. Adding a letter to the right

We follow Ukkonen's algorithm for updating the suffix tree. Recall that Ukkonen's algorithm proceeds by updating the *active node* in the tree. At the beginning of each iteration, the active node corresponds to the longest repeated suffix, i.e. to factor $y[s_i\,..\,i+m-1]$. The node corresponding to the longest repeated prefix is called the *head node*.

The algorithm starts from the active node and updates it following the suffix links until reaching a node with an outgoing edge starting with $y[i+m]$: this node corresponds to the suffix starting at $\tilde{s}_i$. At the same time, we compute MAWs of type 3 that are created. For each $s_i \le j < \tilde{s}_i$, we perform the following.

- If the active node is implicit, it is made explicit. Its set of preceding letters is the set equal to its child's set. The untagged letters of the *maw*-set of its child are moved to the *maw*-set of the active node. The tagged letters of the *maw*-set of its child are untagged. If the last node created at this window shift does not have a suffix link, a suffix link is set from this node to the active node. We add the letter corresponding to this suffix link to the set of preceding letters of the active node.
- We create a leaf labelled j, with $y[j-1]$ in its set of preceding letters. An edge from the active node to this leaf is created with label $y[i+m]$.
- For each letter $a \neq y[j-1]$ in the set of preceding letters of the active node, $ay[s_i+j\,..\,i+m] \in M_{m+1}(i)\backslash M_m(i)$ (type 3 in Lemma 3), therefore a is added to the *maw*-set of the leaf.

The current active node corresponds to the factor $y[\tilde{s}_i\,..\,i+m-1]$. According to Lemma 2, there is exactly one MAW to be deleted which is $y[\tilde{s}_i-1\,..\,i+m]$. This MAW is stored in the child of the active node by following the edge starting with $y[i+m]$; we remove $y[\tilde{s}_i-1]$ (tagged or not) from its *maw*-set.

The active node is then updated by following the edge starting with $y[i+m]$; now it corresponds to the factor $y[\tilde{s}_i\,..\,i+m]$. If the head node was also corresponding to the factor $y[\tilde{s}_i\,..\,i+m-1]$, it is moved down with the active node; we have $\tilde{p}_{i+1} = p_i + 1$, otherwise we have $\tilde{p}_{i+1} = p_i$. If the active node is explicit, its set of preceding letters is updated by adding $y[\tilde{s}_i-1]$.

Then, for each letter b occurring after an occurrence of $y[\tilde{s}_i\,..\,i+m]$ in $y[i\,..\,i+m-1]$, $y[\tilde{s}_i-1\,..\,i+m]b \in M_{m+1}(i)\backslash M_m(i)$ (type 2 in Lemma 3). These MAWs are stored in their corresponding child of the active node. If the active node is implicit, there is only one of them and we tag the letter.

By Lemma 3, if $ss_i = \tilde{s}_i - 1$, then $y[i+m]y[\tilde{s}_i-1\,..\,i+m]$ is the new MAW of type 1. We store it in the *maw*-set of the child of the active node by following the edge starting with $y[i+m]$.

3.2.2. Deleting the leftmost letter

Note that the longest repeated prefix of $y[i\,..\,i+m]$ is $y[i\,..\,\tilde{p}_{i+1}]$, and its longest repeated suffix is $y[\tilde{s}_i\,..\,i+m]$. At the beginning of this step, they correspond respectively to the head node and the active node of the current suffix tree. Consider the parent of the oldest leaf of the tree, similarly to Senft's algorithm two cases are distinguished.

- If the head node is an explicit node, then it is the parent of the oldest leaf. We remove the leaf and its incoming edge. If the head node has only one remaining child, the node is deleted and the two edges are merged, and the *maw*-set associated with the node is added to the leaf.
- Otherwise, the head node occurs on the edge leading to the oldest leaf. We replace the leaf with a new one labelled by $\tilde{s}_i$, with $y[\tilde{s}_i-1]$ as the only preceding letter, and the edge is relabelled with $y[\tilde{s}_i-1]$. $y[\tilde{s}_i-1]$ is added to the set of preceding letters of the parent of the leaf.

The MAWs associated with the leaf we have deleted were those of type 3 (Lemma 6). We now update the tree and compute the other MAWs that have to be removed or added.

We visit the oldest leaf in the tree and empty its set of preceding letters. Then we move up in the tree following back the edges until we have covered $m + i - \tilde{p}_{i+1}$ letters. The head node is set to this node: it corresponds to the factor $y[i+1\,..\,\tilde{p}_{i+1}]$. If the active node was equal to the head node, we move the active node to this node; we have $s_{i+1} = \tilde{s}_i - 1$, otherwise we have $s_{i+1} = \tilde{s}_i$. Each of the explicit nodes visited on the path from the oldest leaf to the head node corresponds to a factor $y[i+1\,..\,j]$, with $p_{i+1} \geq j > \tilde{p}_{i+1}$. For each of them, we remove $y[i]$ from their set of preceding letters. For each of their children, we remove letter $y[i]$ (tagged or not) from their *maw*-set (type 2 Lemma 6).

There is at most one MAW of type 1 that has to be deleted (Lemma 6). It exists if and only if $y[i] = y[i+1]$ and $pp_{i+1} = \tilde{p}_{i+1} + 1$, in which case we remove it from the *maw*-set of the child of the head node by following the edge starting with $y[i]$. According to Lemma 5, removing the leftmost letter creates one MAW, which is $y[i]y[i+1\,..\,\tilde{p}_{i+1}+1]$, thus $y[i]$ is added to the *maw*-set of the child of the head node by following the edge starting with $y[\tilde{p}_{i+1}+1]$. If the head node is implicit and thus equal to the active node, letter $y[i]$ is tagged.

Finally, if the head node is above the parent of the oldest leaf of the tree, we move it down to this node. If the active node is implicit and on the edge leading to the oldest leaf of tree we set the head node equal to the active node.

3.2.3. Correctness

We start with an empty tree, then we first use the operation 'adding a letter to the right', described in paragraph 3.2.1, m times, m being the size of the window. Thus we obtain the suffix tree $T_m(0)$ together with the *maw*-set, the set of preceding letters of each of its nodes, and the set of MAWs $M_m(0)$ of $y[0\,..\,m-1]$. Then each shift of the sliding window across y is decomposed in two steps. First an operation of 'adding a letter to the right' followed by an operation of 'deleting the leftmost letter', described in paragraph 3.2.2. This way we obtain, for each position i, the suffix tree $T_m(i)$ together with the *maw*-set, the set of preceding letters of each of its nodes, and the set of MAWs $M_m(i)$ of $y[i\,..\,i+m-1]$. We now prove the correctness of these two steps. We focus on the update of the *maw*-set of the nodes as this is our main contribution, the rest being similar to Senft's and Ukkonen's algorithms.

Adding a letter to the right We start with the suffix tree and sets corresponding to $y[i\,..\,i+m-1]$ and we update them to obtain those corresponding to $y[i\,..\,i+m]$. As shown in Lemmas 2 and 3, this operation deletes exactly one MAW, $y[\tilde{s}_i\,..\,i+m-1]y[i+m]$, and creates at most $(\tilde{s}_i - s_i)(\sigma-1)+\sigma+1$ MAWs, divided in three types, as illustrated by Fig. 1.

There are at most $(\tilde{s}_i - s_i)(\sigma-1)$ MAWs of type 3, $w = ay[k\,..\,i+m]$, with $a \in \Sigma$ and $s_i \leq k \leq \tilde{s}_i$. Thus by definition of mapping f, all these MAWs are stored in the leaves that are created when traversing the tree from the active node of $y[i\,..\,i+m-1]$ that corresponds to factor $y[s_i\,..\,i+m-1]$ to the node that corresponds to factor $y[\tilde{s}_i\,..\,i+m-1]$.

Once we have reached the node that corresponds to factor $y[\tilde{s}_i\,..\,i+m-1]$, we remove the MAW to be deleted, as it is stored in its child, by following the edge starting with $y[m+1]$. Then we update the active node and the head node if necessary by following the same edge. There at most σ MAWs of type 2, $w = y[\tilde{s}_i - 1\,..\,i+m]b$, with $b \in \Sigma$, and at most one of type 1, $w = y[\tilde{s}_i - 1\,..\,i+m]y[i+m]$. They are stored in the children of the active node of $y[i\,..\,i+m]$, once we have reached it.

During the traversal, we go through all 'new factors', and we also update the set of preceding letters of these nodes, as explained in paragraph 3.2.1. Thus at the end of this step, we obtain the suffix tree $T_{m+1}(i)$ together with the *maw*-set, the set of preceding letters of each of its nodes, and the set of MAWs $M_{m+1}(i)$ of $y[i\,..\,i+m]$.

Deleting the leftmost letter Here we start with the suffix tree and sets corresponding to $y[i\,..\,i+m]$ and we update them to obtain those corresponding to $y[i+1\,..\,i+m]$. This step works the same way as the other, but instead of adding new leaves we remove the oldest. By definition the oldest leaf is a child of the head node (the node that corresponds to the longest repeated prefix $y[i\,..\,\tilde{p}_{i+1}]$). As shown by Lemmas 5 and 6, this operation adds exactly one MAW, $y[i]y[i+1\,..\,\tilde{p}_{i+1}+i]$, and deletes at most $(p_{i+1} - \tilde{p}_{i+1})(\sigma-1)+\sigma+1$ MAWs, divided in three types, as illustrated by Fig. 2.

There are at most σ MAWs of type 3, $w = ay[i\,..\,\tilde{p}_{i+1}+1]$, with $a \in \Sigma$. They are all stored in the child of the head node by following the edge starting with $y[\tilde{p}_{i+1}+1]$; this child is the oldest leaf. So when we remove it, we also remove those MAWs.

Then to update the tree we traverse the tree from the new oldest leaf, that corresponds to $y[i+1\,..\,i+m]$ to the node corresponding to $y[i+1\,..\,\tilde{p}_{i+1}]$. Each of the nodes visited during the traversal corresponds to a factor $y[i+1\,..\,j]$, with $p_{i+j} \geq j \geq \tilde{p}_{i+1}$. There are at most $(p_{i+1} - \tilde{p}_{i+1})(\sigma-1)$ MAWs of type 2, $w = y[i]y[i+1\,..\,j]b$, with $b \in \Sigma$ and $p_{i+j} \geq j \geq \tilde{p}_{i+1}$. Thus they are all stored in the children of the nodes we visit during the traversal.

There is at most one MAW of type 1, $w = y[i]y[i+1\,..\,\tilde{p}_{i+1}]y[i]$. If it exists it is stored in the child of the node corresponding to $y[i+1\,..\,\tilde{p}_{i+1}]$ by following the edge starting with $y[i]$. The MAW to be created, $w = y[i]y[i+1\,..\,\tilde{p}_{i+1}+i]$, is also stored in a child of the node corresponding to $y[i+1\,..\,\tilde{p}_{i+1}]$.

Thus during the traversal we go through all the factors that are not preceded anymore by $y[i]$ and we also update their set of preceding letters, as explained in paragraph 3.2.2. Finally we obtain the suffix tree $T_m(i+1)$ together with the *maw*-set, the set of preceding letters of each of its nodes, and the set of MAWs $M_m(i+1)$ of $y[i+1\,..\,i+m]$.

3.2.4. Complexity

The algorithm extends Senft's algorithm for the construction of the suffix tree in a sliding window. For both addition and deletion of a letter, the number of operations is $\mathcal{O}(\sigma(\tilde{s}_i - s_i))$ and $\mathcal{O}(\sigma(p_{i+1} - \tilde{p}_{i+1}))$. Similar to the proof of Theorem 8,

we obtain that the total number of operations is $\mathcal{O}(\sigma n)$. We use $\mathcal{O}(\sigma m)$ space to store the suffix tree for the window. The σ factor comes from storing an array of size σ at each explicit node for constant-time child queries. We also use up to $4m$ arrays of size σ each to store the two sets of letters. We also store the word itself over two windows. Thus the total space complexity is bounded by $\mathcal{O}(\sigma m)$. We thus obtain our main result.

Theorem 10. *Given a word of length n over an alphabet of size σ, our algorithm computes the set of MAWs in a sliding window of size m in $\mathcal{O}(\sigma n)$ time and $\mathcal{O}(\sigma m)$ space.*

4. An illustrative example

This section illustrates a few steps of our algorithm run on a simple example. The target word is $y=\overset{0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11}{\mathsf{ACACAAGCAGAA}}\ldots$ and the window length is $m = 8$.

The first three positions of the window and the associated shifts are illustrated in Figs. 3–6. The longest repeated prefixes (head node) are shown in green and the longest repeated suffixes (active node) are shown in blue. The modifications from one tree to the other are shown in red. The sets representing the MAWs are denoted by M and those representing the set of preceding letters are denoted by B. Tagged letters are shown by a subscript '$_+$'. The figures are to be read from left to right and then from top to bottom.

First position of the window: $z_0 = \overset{0\,1\,2\,3\,4\,5\,6\,7}{\mathsf{ACACAAGC}}$ followed by A (see Fig. 3).

- **Part 1**: Appending A to the right. The active node has a child by A, thus there is no MAWs of type 3, the active node is also the node corresponding to the suffix starting at $\tilde{s}_0$.
  - We remove the MAW GCA, stored in the destination of the edge containing the active node.
  - We move down the active node (in blue) of one letter, this node is explicit. We update its set of letters by adding the letter G.
  - In each child of the active node we store $y[\tilde{s}_0 - 1] = \mathsf{G}$ in their *maw*-set (type 2).
  - $ss_0 \neq \tilde{s}_0 - 1$ thus there is no MAW of type 1.
- **Part 2**: Removing A from the left.
  - The head node corresponds to an explicit node in green. We remove the MAWs stored in the oldest leaf of the tree (in red), CACAC (type 3). The head node has only one remaining child. Thus we make it implicit, and its *maw*-set goes to its leaf. We add its set of letters to its parents.
  - We go to the oldest leaf in the tree, we empty its set of preceding letters. Then we move 6 letters up. We move the head node to this node. There were no explicit nodes on the way, thus there is no MAW of type 2.
  - $pp_1 \neq \tilde{p}_1 + 1$, thus there is no MAW of type 1.
  - The MAW to create is ACAC, we store it in the oldest leaf with the letter A.

Second position of the window: $z_1 = \overset{1\,2\,3\,4\,5\,6\,7\,8}{\mathsf{CACAAGCA}}$ followed by G (see Fig. 4).

- **Part 1**: Appending G to the right.
  - the active node is explicit but it does not have a child by G, thus we create a leaf labelled $s_1 = 7$, with $y[s_1 - 1] = \mathsf{G}$ as a preceding letter. For every letter a preceding the active node, except G, we add a as a MAW in the leaf, thus we add A. We move up the active node following the suffix link.
  - the active node is explicit and it has a child by G.
    - We remove the MAW CAG stored in the child of the active node by G.
    - We move down the active node by following G.
    - The active node is now implicit, thus there is only one letter following its corresponding factor, we add $y[\tilde{s}_1 - 1] =$ C in the *maw*-set of the destination of its edge (type 2), we tag this letter.
    - $ss_1 \neq \tilde{s}_1 - 1$ thus there is no MAW of type 1.
- **Part 2**: Removing C from the left.
  - The head node is explicit, we remove the oldest leaf, and its corresponding MAWs ACAC and GCAC (type 3).
  - We visit the oldest leaf of tree. We empty its set of preceding letters. Then we move up 7 letters. We move the head node to this node. There were no explicit nodes on the way, thus there is no MAW of type 2.
  - $pp_2 \neq \tilde{p}_2 + 1$ this there is no MAW of type 1.
  - There is one MAW to create it is $y[1]y[2..\tilde{p}_2 + 1] =$ CAC we add it to the head node.

Third position of the window: $z_2 = \overset{2\,3\,4\,5\,6\,7\,8\,9}{\mathsf{ACAAGCAG}}$ followed by A.

- **Part 1**: Appending A to the right (see Fig. 5).
  - The active node is implicit, we make it explicit. We set its set of preceding letters equal to its child's. There is no untagged letter in the *maw*-set of its child, we untag the tagged letter of its child.

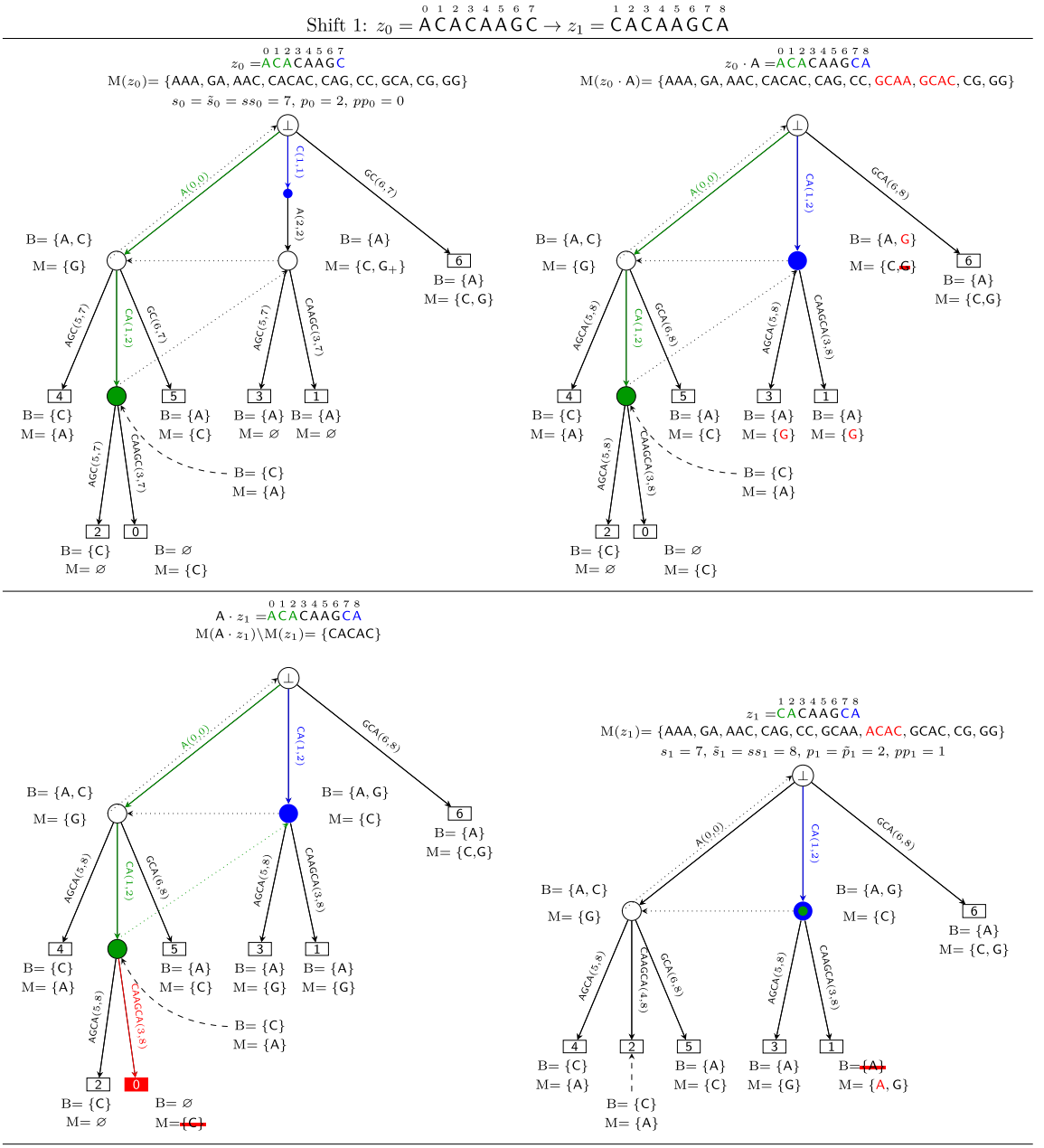

Fig. 3. First shift of the window.

- We create a new leaf, labelled $s_2 = 8$ with $y[s_2 - 1] = \mathsf{C}$ in its set of preceding letters.
- $\mathsf{A} \neq y[s_2 - 1]$ precedes the active node, thus we add A in the *maw*-set of the new leaf (type 3). And we move the active node following the suffix links.

– The active node is still implicit, we make it explicit. We set its set of preceding letters equal to its child's. We move the untagged letters of the *maw*-set of its child to the *maw*-set of the active node.
- We create a new leaf, labelled $s_2 + 1 = 9$ with $y[s_2] = \mathsf{A}$ in its set of preceding letters.
- There is no MAWs of type 3, as the active node is only preceded by A. We create a suffix link from the last node created to the active node. We move up the active node by following the suffix links.

– The active node is now explicit, it is the root:
- We remove $y[\tilde{s}_2 - 1] = \mathsf{G}$ stored in the child of the active node by A.
- We move down the active node by following A. This node is explicit, we add $y[\tilde{s}_2 - 1] = \mathsf{G}$ to its set of preceding letters.

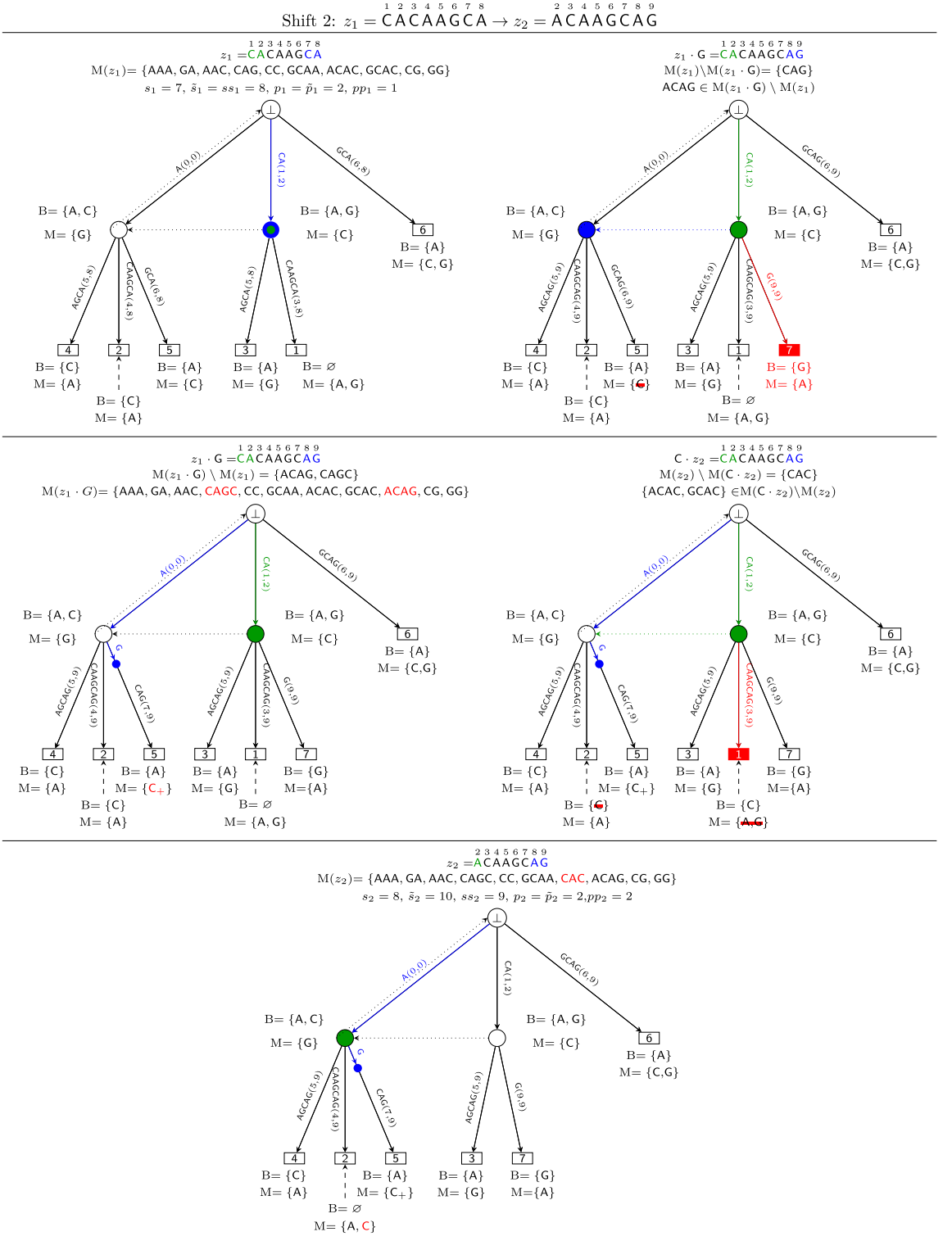

Fig. 4. Second shift of the window.

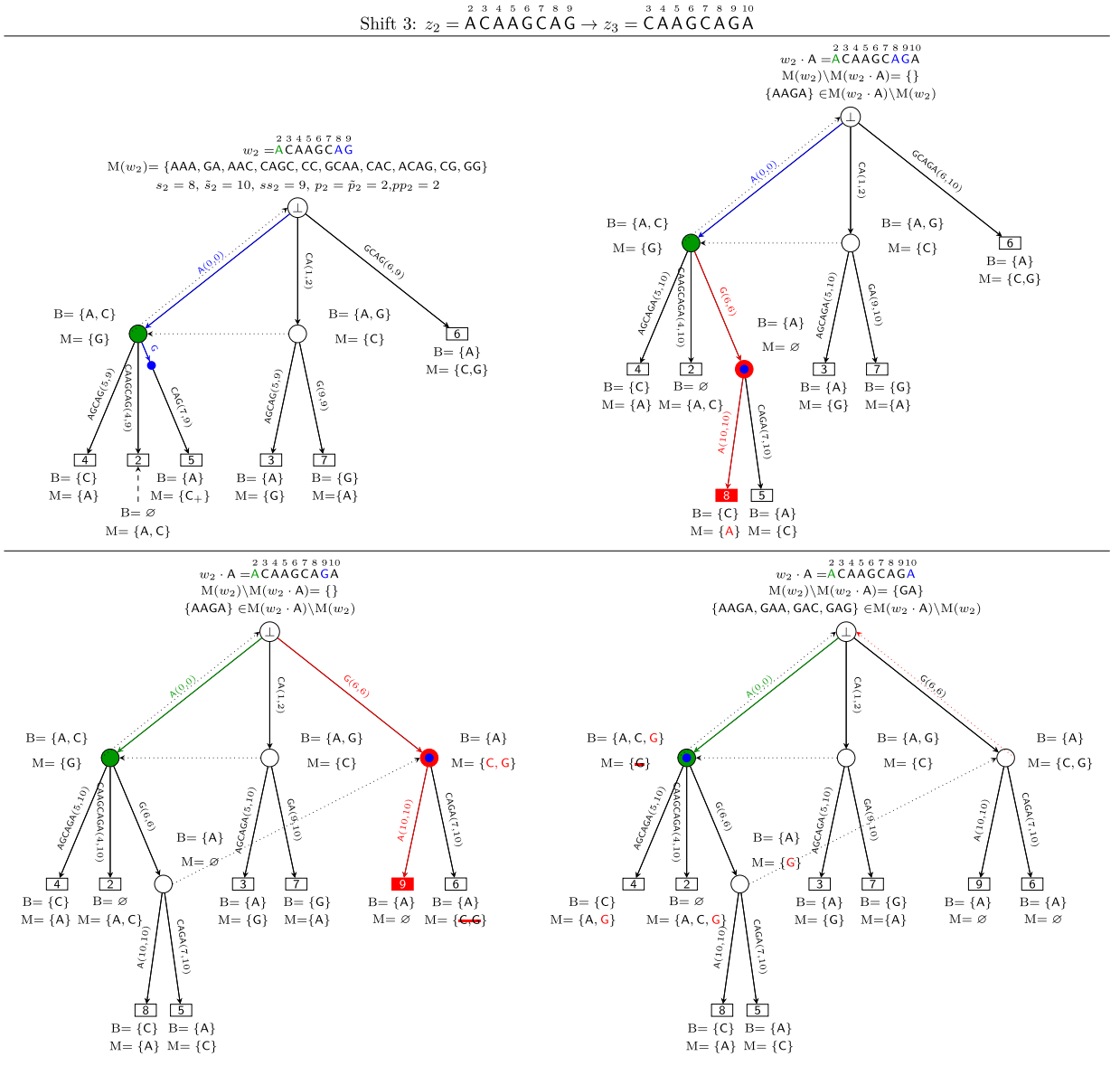

Fig. 5. Third shift of the window (Part 1).

  - We add $y[\tilde{s}_2 - 1] = \mathsf{G}$ in the *maw*-set of each child of the active node (type 2).
  - There is no MAWs of type 1.
- **Part 2**: Removing A from the left (see Fig. 6).
  - The head node is explicit, we remove the oldest leaf, and its corresponding MAWs AAC, GAC and GAC (type 3).
  - We visit the oldest leaf of tree. we empty its set of preceding letters. Then we move 8 letters up. We move the head node to this node. There is one explicit node on the way, we consider it.
    - We remove $y[2] = \mathsf{A}$ from its set of preceding letters.
    - We remove $y[2] = \mathsf{A}$ from the sets of MAWs of its children.
  - There is no MAWs of type 1, there is one MAW to add it is AC, we store it the child of the head node that lead to the oldest leaf in the tree.
  - We move the head node to the node corresponding to the parent of the oldest leaf in the tree.

5. Application to online approximate pattern matching

As a consequence of Theorem 10, we obtain the following result.

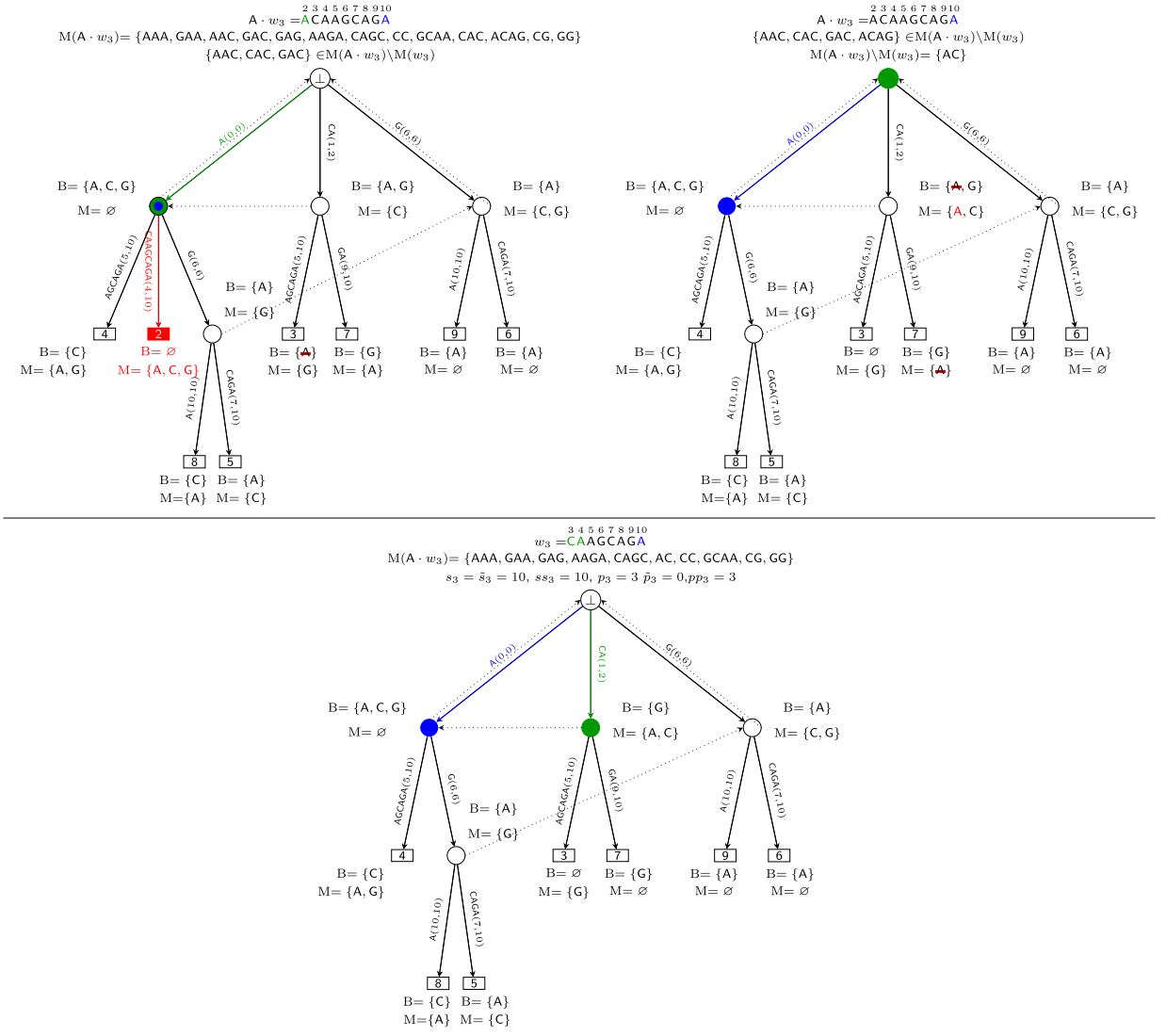

Fig. 6. Third shift of the window (Part 2).

Theorem 11. *Given words x of length m and y of length $n \geq m$ over an alphabet of size σ, all distances LWI$(x, y[i\,..\,i+m-1])$, for $0 \leq i \leq n-m$, can be computed online in $\mathcal{O}(\sigma n)$ time and $\mathcal{O}(\sigma m)$ space. In particular, for any distance threshold d, all occurrences i verifying LWI$(x, y[i\,..\,i+m-1]) \leq d$ can be found within the same complexity bounds.*

Proof. As a pre-processing step, we build the suffix tree of x and compute the MAWs of x. At the same time, by Lemma 9, we represent all MAWs of x by storing a set of letters in each explicit node of the tree. This can be done in $\mathcal{O}(\sigma m)$ time and space [13].

We then apply Theorem 10 to build the suffix tree for a sliding window of size m over y on top of the suffix tree of x. This way when a MAW w is created or deleted we can update the LWI score in $\mathcal{O}(1)$ time as we can check if it is a MAW of x or not. If w is created and $w \notin M(x)$ we add $\frac{1}{|w|^2}$ to the LWI score; if w is created and $w \in M(x)$ we add nothing. If w is deleted and $w \notin M(x)$ we add $-\frac{1}{|w|^2}$ to the LWI score; if w is deleted and $w \in M(x)$ we add nothing. This concludes the proof. □

Declaration of competing interest

None.

References

[1] Y. Almirantis, P. Charalampopoulos, J. Gao, C.S. Iliopoulos, M. Mohamed, S.P. Pissis, D. Polychronopoulos, On avoided words, absent words, and their application to biological sequence analysis, Algorithms Mol. Biol. 12 (1) (2017) 5.
[2] C. Barton, A. Héliou, L. Mouchard, S.P. Pissis, Linear-time computation of minimal absent words using suffix array, BMC Bioinform. 15 (11) (2014).
[3] C. Barton, A. Héliou, L. Mouchard, S.P. Pissis, Parallelising the computation of minimal absent words, in: Parallel Processing and Applied Mathematics: 11th International Conference, Part II, in: Lecture Notes in Computer Science, vol. 9574, Springer, 2015, pp. 243–253.
[4] M. Béal, F. Mignosi, A. Restivo, Minimal forbidden words and symbolic dynamics, in: 13th Annual Symposium on Theoretical Aspects of Computer Science, in: Lecture Notes in Computer Science, vol. 1046, Springer, Grenoble, France, 1996, pp. 555–566.
[5] D. Belazzougui, F. Cunial, J. Kärkkäinen, V. Mäkinen, Versatile succinct representations of the bidirectional Burrows-Wheeler transform, in: 21st Annual European Symposium on Algorithms, Sophia Antipolis, France, in: Lecture Notes in Computer Science, vol. 8125, Springer, 2013, pp. 133–144.
[6] S. Chairungsee, M. Crochemore, Using minimal absent words to build phylogeny, Theor. Comput. Sci. 450 (2012) 109–116.
[7] P. Charalampopoulos, M. Crochemore, G. Fici, R. Mercas, S.P. Pissis, Alignment-free sequence comparison using absent words, Inf. Comput. 262 (2018) 57–68.
[8] P. Charalampopoulos, M. Crochemore, S.P. Pissis, On extended special factors of a word, in: T. Gagie, A. Moffat, G. Navarro, E. Cuadros-Vargas (Eds.), String Processing and Information Retrieval – Proceedings of the 25th International Symposium, SPIRE 2018, Lima, Peru, October 9–11, 2018, in: Lecture Notes in Computer Science, vol. 11147, Springer, 2018, pp. 131–138.
[9] T. Crawford, G. Badkobeh, D. Lewis, Searching page-images of early music scanned with OMR: a scalable solution using minimal absent words, in: E. Gómez, X. Hu, E. Humphrey, E. Benetos (Eds.), Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, September 23–27, 2018, 2018, pp. 233–239.
[10] M. Crochemore, G. Fici, R. Mercas, S.P. Pissis, Linear-time sequence comparison using minimal absent words, in: 12th Latin American Symposium on Theoretical Informatics (LATIN), Ensenada, Mexico, in: Lecture Notes in Computer Science, vol. 9644, Springer, 2016, pp. 334–346.
[11] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[12] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S.P. Pissis, Y. Ramusat, Minimal absent words in a sliding window and applications to on-line pattern matching, in: 21st International Symposium (FCT), Bordeaux, France, in: Lecture Notes in Computer Science, vol. 10472, Springer, Berlin Heidelberg, 2017, pp. 164–176.
[13] M. Crochemore, F. Mignosi, A. Restivo, Automata and forbidden words, Inf. Process. Lett. 67 (3) (1998) 111–117.
[14] M. Crochemore, F. Mignosi, A. Restivo, S. Salemi, Data compression using antidictionaries, Proc. IEEE 88 (11) (2000) 1756–1768.
[15] G. Fici, Minimal Forbidden Words and Applications, Phd thesis, Université de Marne la Vallée, 2006.
[16] Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, M. Takeda, Computing DAWGs and minimal absent words in linear time for integer alphabets, in: 41st International Symposium on Mathematical Foundations of Computer Science, in: LIPIcs, vol. 58, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, p. 38.
[17] H. Fukae, T. Ota, H. Morita, On fast and memory-efficient construction of an antidictionary array, in: Proceedings of the 2012 IEEE International Symposium on Information Theory, ISIT 2012, Cambridge, MA, USA, July 1–6, 2012, IEEE, 2012, pp. 1092–1096.
[18] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[19] G. Hampikian, T.L. Andersen, Absent sequences: nullomers and primes, in: Pacific Symposium on Biocomputing (PCB), Maui, Hawaii, World Scientific, 2007, pp. 355–366.
[20] A. Héliou, S.P. Pissis, S.J. Puglisi, emMAW: Computing minimal absent words in external memory, Bioinformatics 33 (2017) 2746–2749.
[21] J. Herold, S. Kurtz, R. Giegerich, Efficient computation of absent words in genomic sequences, BMC Bioinform. 9 (2008) 167.
[22] F. Mignosi, A. Restivo, M. Sciortino, Words and forbidden factors, Theor. Comput. Sci. 273 (1–2) (2002) 99–117.
[23] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.
[24] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2008.
[25] T. Ota, H. Fukae, H. Morita, Dynamic construction of an antidictionary with linear complexity, Theor. Comput. Sci. 526 (2014) 108–119.
[26] T. Ota, H. Morita, On the construction of an antidictionary with linear complexity using the suffix tree, IEICE Trans. A 90 (11) (2007) 2533–2539.
[27] T. Ota, H. Morita, On a universal antidictionary coding for stationary ergodic sources with finite alphabet, in: International Symposium on Information Theory and Its Applications (ISITA), IEEE, 2014, pp. 294–298.
[28] M.S. Rahman, A. Alatabbi, T. Athar, M. Crochemore, M.S. Rahman, Absent words and the (dis)similarity analysis of DNA sequences: an experimental study, BMC Bioinform. Notes 9 (1) (2016) 1–8.
[29] M. Senft, Lossless Data Compression Using Suffix Trees, Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic, 2003.
[30] M. Senft, Suffix tree for a sliding window: an overview, in: WDS, Matfyzpress, 2005, pp. 41–46.
[31] R.M. Silva, D. Pratas, L. Castro, A.J. Pinho, P.J.S.G. Ferreira, Three minimal sequences found in Ebola virus genomes and absent from human DNA, Bioinformatics 31 (15) (2015) 2421–2425.
[32] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[33] Z. Wu, T. Jiang, W. Su, Efficient computation of shortest absent words in a genomic sequence, Inf. Process. Lett. 110 (14–15) (2010) 596–601.